Environmental Monitoring and Pollution Prediction System

Nouman Amjad

December 15, 2024

Contents

Introduction

Overview of the Project

This project builds a robust MLOps pipeline for monitoring environmental parameters, predicting pollution levels, and visualizing performance metrics. It incorporates tools such as:

- **DVC (Data Version Control)**: Ensures reproducibility and management of collected data.
- **MLflow**: Tracks model experiments and manages the lifecycle of machine learning models.
- **Prometheus and Grafana**: Monitors system metrics and visualizes performance dashboards.

The pipeline integrates continuous data collection, model training, deployment, and monitoring to automate the pollution trend prediction process.

Structure of the Documentation

The report is organized into the following tasks:

1. Task 1: Collecting and managing data using DVC.
2. Task 2: Building, deploying, and tracking the LSTM model using MLflow.
3. Task 3: Monitoring the deployed model and optimizing performance with Prometheus and Grafana.

Chapter 1

Task 1: Managing Environmental Data with DVC

1.1 1.1 Introduction to Data Collection

Environmental data includes critical metrics such as:

- Air Quality Index (AQI)
- Pollutants (e.g., CO, NO_2, PM2.5, PM10, Ozone)
- Weather Conditions (Temperature, Humidity, Wind Speed)

The data is collected using the OpenWeatherMap API for real-time weather and pollution updates.

1.2 1.2 Setting Up the DVC Repository

Why use DVC?

- **Data versioning**: Tracks changes to datasets similar to Git.
- **Storage optimization**: Ensures efficient data storage.
- **Reproducibility**: Maintains consistency for model training.

To set up DVC:

```
# Initialize Git and DVC
git init
dvc init
```

Listing 1.1: Initialize DVC

1.3 1.3 Configuring Remote Storage

For managing data remotely, configure Google Drive as DVC storage:

```
dvc remote add -d myremote gdrive://YOUR_DRIVE_ID
```

Listing 1.2: Configure DVC Remote

Here, `YOUR_DRIVE_ID` refers to the folder ID on Google Drive.

1.4 1.4 Data Fetching Script

The Python script fetches data from the OpenWeatherMap API:

- Air Pollution Data: Measures AQI and pollutant levels.
- Weather Data: Captures temperature, humidity, and pressure.

```python
import requests
import pandas as pd
from datetime import datetime
import os
from dotenv import load_dotenv

# Load API Key
load_dotenv()
API_KEY = os.getenv("API_KEY")
lat, lon = 33.6938, 73.0651  # Coordinates for Islamabad

# Function to fetch data
def fetch_data():
    url = f"http://api.openweathermap.org/data/2.5/weather?lat={lat}&
    lon={lon}&appid={API_KEY}"
    response = requests.get(url)
    return response.json()

data = fetch_data()
df = pd.DataFrame([data])
df.to_csv("Data/environmental_data.csv", index=False)
```

Listing 1.3: Data Fetch Script

1.5 1.5 Data Versioning and Automation

To track the collected data:

```
dvc add Data/environmental_data.csv
dvc commit -m "Added environmental data"
dvc push
```

Listing 1.4: Add Data to DVC

Automation: Use a cron job to schedule data fetching every hour:

```
crontab -e
0 * * * * /usr/bin/python3 /path/to/data_fetch.py
```

Listing 1.5: Set Up Cron Job

—

Chapter 2

Task 2: Developing and Deploying the LSTM Model

2.1 2.1 Data Preparation

The environmental data is cleaned and normalized:

- Missing values: Imputed using the mean of the dataset.
- Normalization: Scales data to ensure effective model training.

2.2 2.2 Building the LSTM Model

The LSTM model architecture:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(5, 5)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

Listing 2.1: LSTM Model Architecture

2.3 2.3 Tracking with MLflow

Log experiments, metrics, and parameters:

```
import mlflow

mlflow.start_run()
mlflow.log_param("batch_size", 32)
mlflow.log_metric("mae", 0.23)
mlflow.end_run()
```

Listing 2.2: MLflow Tracking

2.4 2.4 Model Deployment

Deploy the model as a Flask API:

```python
from flask import Flask, request, jsonify
from tensorflow.keras.models import load_model
import numpy as np

app = Flask(__name__)
model = load_model("models/lstm_model.h5")

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json
    features = np.array(data["features"]).reshape(1, 5, 5)
    prediction = model.predict(features)
    return jsonify({"prediction": prediction.tolist()})
```

Listing 2.3: Deploying Model

—

Chapter 3

Task 3: Monitoring and Optimization

3.1 3.1 Monitoring System Metrics

Prometheus is configured to monitor:

- API request count
- Response latency
- Error rates

Prometheus configuration:

```yaml
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: "flask_api"
    static_configs:
      - targets: ["localhost:8000"]
```

Listing 3.1: Prometheus Scrape Config

3.2 3.2 Visualizing Metrics in Grafana

Steps to set up Grafana:

- Add Prometheus as a data source.
- Create panels to display:
  - Prediction Requests
  - Response Latency
  - Model Predictions

3.3 3.3 Live Data Testing and Optimization

- Fetch live data continuously.
- Analyze predictions and evaluate model performance.
- Optimize system performance by tuning model parameters and API efficiency.

—

Conclusion

The project successfully integrates real-time data collection, predictive modeling, and system monitoring using modern MLOps tools. Future work includes refining models with additional features and scaling the pipeline for larger datasets.